Bachelor's Thesis

Building a Massively Multiplayer Online Role-playing games in small scale

Timo Nevalainen

Information Technology

2009

TURKU UNIVERSITY OF APPLIED SCIENCES

ABSTRACT OF THESIS

Degree Programme: Information Technology	
Author(s): Timo Nevalainen	
Title: Building a Massively Multiplayer Online Role-playing games in small scale	
Specialization line: Software Engineering	Instructor(s): Janne Röslöf, D. Sc
Date: September 2009	Total number of pages: 50

The project is aimed to get a view of how to build basic elements required in a Massively Multiplayer role-playing games. Specifically concentrating on the issues of Graphical User Interface, 3D engine and network communication. It doesn't concern itself with game design as such like game balance and level design. Though somewhat limited in its scope, it manages to provide a reasonable starting point from where to expand further. The program was built from scratch with a minimal amount of 3rd party libraries and those used were mainly for low level activity. This thesis views network model, how to set up a world, characters filling the world, a graphical user interface and a 3D graphic engine.

Some thought has been put in separating parts of code from each other giving independent parts which can be reused in other programs with minimal fuss. A prime example of this is the graphical user interface which requires very little work to utilize in other programs.

Keywords: network, MMORPG, technical, design

Deposit at: Library of Turku University of Applied Sciences

TURUN AMMATTIKORKEAKOULU

OPINNÄYTETYÖN TIIVISTELMÄ

Koulutusohjelma: Tietotekniikan koulutusohjelma	
Tekijä: Timo Nevalainen	
Työn nimi: Asiakas-palvelinverkkosovellus	
Suuntautumisvaihtoehto: Ohjelmisto- ja järjestelmätekniikka	Ohjaaja(t): Janne Röslöf, Tkt
Aika: Syyskuu 2009	Sivumäärä: 50

Projektin tarkoituksena on tutustua keskeisiin elementteihin, jotka muodostavat yhdessä verkon yli pelattavan roolipelin, kuten Ultima Online ja World of Warcraft. Erityisesti huomion kohteena on 3D-grafiikkamoottori, käyttöliittymä ja verkkoliikenne.

Esimerkkisovellus on luotu C++ - kielellä käyttäen mahdollisimman pientä määrää kolmannen osapuolten kirjastoja. Ohjelma katsoo, kuinka verkkoliikenne toimii, miten rakentaa graaffinen käyttöliittymä käyttäjälle, maailman luomista ja 3d-grafiikkamoottorin

Merkittävä tavoite ohjelmakoodissa on ollut pyrkimys modulaarisuuteen mahdollistaen saman koodin uudelleenhyödyntämistä toisissa projekteissa ilman kattavaa uudelleentoteuttamista. Erityisen hyvä esimerkki tästä on käyttöliittymästä vastaava osa, joka on liitettävissä melkein sellaisenaan muihinkin openGL-pohjaisiin sovelluksiin.

Kaikenkaikkiaan ohjelma pyrkii näyttämään, kuinka rakentaa asiakas-palvelinsovelluksen erityisesti verkkopelien kohdalla.

Hakusanat: verkkosovellus, MMORPG, tekninen, suunnittelu

Säilytyspaikka: Turun ammattikorkeakoulun kirjasto

Contents

Notation

SDL	Simple Directmedia Library
OpenGL	Open Graphics Library
SDL_Net	Simple Directmedia Library net
MMORPG	Massively Multiplayer Online Role-Playing Game
GUI	Graphical User Interface

1 Introduction

This project is an attempt to create a small scale version of a working server-client program which would be suitable for network based game in style of Massively multiplayer online role-playing games (henceforth shortened as MMORPG) like the Ultima Online and the World of Warcraft. Though creating such a game is beyond scope of any individual person it should be possible to create small scale mockup highlighting several of design aspects involved in creation of such a program. This isn't near being powerful enough for public use even if all the game features would be implemented

To demonstrate the issues involved in the project a simple server and client programs were created for a Linux environment with the mix of C/C++. Both use free cross-platform library SDL_Net (Simple Directmedia Layer Network) for the network connection. In addition the client uses SDL for basic screen setup and event handling, openGL (Open Graphic Library) for drawing graphics, especially 3D, and ftgl (freetype graphic library) for drawing text in openGL application. Rest of the code has been written from scratch.

The server is responsible for keeping track of the game world and its entities while client displays the result for clients. For the communication between clients and server TCP protocol has been chosen largely for its ease of use. In full blown application it would however require pretty nifty technical solution to keep the speed playable. However as certain MMORPG's, most notably Ultima Online and World of Warcraft have shown it is possible to have MMORPG with playable speed with TCP. In most cases though UDP with software implemented safety measures for critical packets would be better choice.

Several design aspects caused particular issues during the development process. In particular limiting the amount of messages each client receives was particularly important for performance reasons. The graphical user interface for the client proved also to be particularly tricky. Especially when decision was made to make it as easily ported to other projects as possible. Graphics engine was major stumbling block and one which isn't near fully done yet, especially when it comes to animation which currently eats way too much memory to be used for large world.

2. History of massively multiplayer online games

Multiplayer network games could be said to have began with the MUD games on the early bulleting board systems. Text based and with limited number of users they couldn't be termed as massive by any stretch of imagination but paved the way for larger games. The major jump toward modern MMORPG's came with the popularization of Internet which opened the concept of networks to common people.

Early attempts like Neverwinter Nights launched on 1991 made good effort and had reasonable success but first commercial big hit was the Ultima Online launched at 1997. Based on the at then still popular world of Ultima role-playing games created by Richard Garriot the game allowed players to explore the world of Britannia with freedom that was quite staggering at the time. Not only limited to being arch typical warriors and mages you could be blacksmith, tailor and even a beggar to name but a few. With the Ultima Online Garriot could say to have achieved his goal of creating virtual world with nearly limitless freedom.

Picture 2.1. Ultima Online. One of the first successes in the field of MMORPG's.

With Ultima Online paving the way for commercial big hits other games like it followed in suit. Everquest, Runescape and Eve Online were some of the titles that gained popularity. But it was with World of Warcraft in 2004 that really blew the bank so to speak. In similar way to Ultima Online it was based on already existing world, this time from the Warcraft series of real time strategy games. By then 3D MMORPG's were starting to appear and WoW followed in suit. Concentrating on the combat system it made several changes in how it was handled so as to make it easier for casual gamers to approach. For example rather than heavy penalties for dying as was usual WoW made it quick and easy to get back in action. In Ultima Online if you died, without another player around to resurrect you instantly, you most likely were going to lose all your equipment for a start and have to take a long walk to find healer to resurrect you. In the World of Warcraft you would be teleported instantly to the healer and you could either resurrect right there with some penalties or move, at faster rate than alive, to where you died with an arrow showing the direction and in either case having your equipment still with you rather than stolen by another player.

Picture 4.2. World of Warcraft. The most successful MMORPG so far.

These changes were in part turning World of Warcraft into hugely successful product surpassing all others by far in terms of subscriptions and revenue. In these days the game has over 10 million subscribers across the world. If you ask about online role-playing games from average person he will almost certainly think of World of Warcraft.

2.1 Feature comparison between Ultima Online and World of Warcraft

The two games vary wildly between them starting from basic premise. Ultima Online is an attempt to create virtual world of Britannia where players can interact between themselves and world. From the scratch it's clear that goal has been to give as much freedom as was possible for the players to play in a way they prefer. World of Warcraft meanwhile decided to concentrate on combat and be good at it in a way that would make it attractive to larger audience.

One big difference between the two are how characters progress and since character progress is such a vital element for games it creates stark contrast between the two. In Ultima Online you have many different skills, some related to combat and some not, which you can improve individually by using them. So if your character fights around a lot with sword he'll become good swordfighter. Spend your time building tables and chairs and you'll become carpenter. Do bit of both and you'll be combination. Only maximum amount of skill points you have limits and if you wanted to change your skills you could set some skill to drop giving room for new one. This results in very flexible build.

World of Warcraft meanwhile uses level system which is common in many pen-and-paper role playing games. As you get experience, mainly from killing monsters with quests providing some as well, you level up which results in higher abilities. World of Warcraft also has skill levels but they are capped by level you have and with no maximum level. You could be good with swords, maces, axes, unarmed and archers all without hitting limits. As they also gain very quickly they aren't that comparable with Ultima Online skills. There are also non-combat skills like blacksmithing but again maximum cap depends on level you are which in turn requires experience which you don't get by blacksmithing so again you have to be warrior. As such they are clearly more of sideline and intended more for players to create high level items for combats than having professions like Ultima Online.

In Ultima Online player versus player combat was originally very brutal and free. If you weren't within guard zone you could be attacked by anybody at will and if you died your equipment was available for looting. This made the world very hard place to live in with death constant danger. Later on splitting world into two different realms (Trammel and Felucca) gave players realm where non-consensual player vs. player combat wasn't allowed (incidentally this was met both by praise and criticism. Some people feel this went too far and hurt the game). World of Warcraft had similar realm system from get-go. There are player vs. player servers and player vs. environment servers where you can only fight against other players if both players are willing by either issuing a duel, entering battlegrounds which are player versus player combat mini-games or by one player attacking village controlled by opposing side making him free to attack target and anybody attacking him is fair play for him. Combined with lesser penalties for death in World of Warcraft (your items only wear down a bit which costs bit of in-game money to repair and time spent on moving from healer to body) means that player vs player combats aren't as much of a worry as they were in Ultima Online originally (and in some free servers where Trammel rules aren't in place).

2.2 Server emulation

Interesting phenomenon has risen in wake of commercial MMORPG's which is server emulation. Rather than building MMORPG from scratch programs have been created through reverse-engineering that reads the original data and responds to clients in correct way. End result is servers that are run by amateurs and which can be connected with normal clients. This has resulted in large number of free servers for popular games. This has been particularly common in Ultima Online after many players were disappointed in changes brought in Age of Shadows supplement. These free servers range anywhere from close approximation of original servers to completely different worlds with custom built map, like Ultima Online free shard called Endor. There have been several Finnish Ultima Online shards as well. Currently there runs one called Northern Winds which currently is based on official world with different time line though in future will become custom one by itself.

Though using clients to connect to these free shards for playing is in violation of terms of service it hasn't stopped them from becoming popular choice. Owners of Ultima Online

however don't pursue seriously these servers as long as they are non-profit. World of Warcraft has some form of measures to limit this as it can track whether client has been used to connect unofficial servers or not.

Another use for these server emulators has been in creation of cartoons like Imanewbie for Ultima Online. [5.]

2.3 Designing the project

The project was started with Ultima Online as major inspiration. The game would be skill based with viewpoint similar to Ultima Online (isometric, i.e. from up to down instead of from behind like World of Warcraft) though players would have ability to rotate the camera like in World of Warcraft. As Ultima Online's mouse driven movement causes problems with using items while movement would be handled by keyboard in similar way to first person shooters. This however has disadvantage when it comes to chat system as you can't type text while moving but mouse is more needed for non-movement purposes than chatting is while moving.

Interface is somewhat limited in that suitable system for detecting double click isn't implemented as SDL has no direct support for it. For this reason most of the job of double click has been allocated to right mouse button. Using items for example happens by right clicking intended item. Similarly choosing target for attacks would happen with right click. Dragging happens with left mouse button as usual. If camera would be allowed to rotate freely that would happen with left button as well by doing it somewhere in which there are no items that would be dragged beneath the cursor.

Main issue to be resolved here is closing windows. In the Ultima online the windows that popped up during game play were closed with right button click. Here that would cause accidental closings of windows which could be lethal during heavy combat. Here we have in essence two feasible choices. Implement the double click and switch target selection and item use for that and in essence recreate interface from Ultima Online or add some sort of a close button to the windows. Technically that isn't too hard to implement but it might be better to implement double click as fast pace common in MMORPGs might mean that life and death could be decided by whether you manage to close the window quickly enough.

As moving mouse to close button is slower than clicking right button on any free space of window this could be decisive.

3. Common design decisions

From the get-go basic premise for the program was client-server architecture with server providing service for more than one client. Clients should be able to interact with each other by chat text if nothing else and the world should be drawn with 3D graphics. Worlds should be customizable and extendable. Attention should be paid so that code isn't hardwired so badly that additional features are hard to add later. Network code would be done with either TCP or UDP with minimal abstraction between ports and application so very high level libraries would be avoided.

3.1 3rd party libraries

While most of the code has been written from scratch there are several libraries that handle low-level stuff which have been used in this program. When deciding which libraries to use priorities were cross-platform and ease of use. Cross-platform capability was something that received particular attention as development was done on Linux environment with goal of porting it to windows in future. As a result several libraries were abandoned as they were Windows specific.

First priority was choice of graphic library for creating the 3D graphics. While technically possible to code sufficient abilities from scratch accessing graphic cards directly this is hardly feasible. As such openGL was chosen due to it being cross-platform, reasonably easy to use and with plenty of tutorials and example material available. Its main competitor DirectX is windows-specific with vastly different syntax which I have never been comfortable with. As such openGL was easy choice as the graphic library.

Another useful library is Simple Direct media Layer which is used to create the windows into which graphic is drawn as well as handles input from mouse and keyboard. There's also another library that is part of SDL which is SDL_Net that has been added to project as well. This provides higher layer access to TCP/UDP protocols in cross-platform library. Both SDL itself and SDL_Net provide layer to operating system specific way to do required things. For example when drawing graphics SDL uses directX when in Windows and XLib in Linux and other systems using X11. This makes porting programs between systems supporting SDL smooth for parts governed by SDL.

Final library used is FTGL which itself uses Freetype library itself. OpenGL itself doesn't have any support whatsoever for drawing text so you have to create your own method, mostly involving creating rectangles with font drawn as texture. FTGL makes this very easy and allows use of existing truetype font files removing the need of drawing font library of your own. As it is fairly easy to use with high number of features it provides more than sufficient amount of features for the rather basic needs of this project.

3.2 TCP or UDP

There are many network protocols but the two most commonly used for network applications like this are TCP and UDP. The differences between two are speed and reliability. Whereupon TCP guarantees that packets arrive to recipient and in correct order UDP makes no such guarantees. However where UDP loses in reliability it gains in speed. When program sends TCP message it will wait for acknowledgment from client. If it doesn't get one in predetermined time or receives acknowledgement of only partial data, due to rest being corrupted, it retransmits part or all it has sent. For web browsing or other such activity where speed isn't critical this is fine and guaranteed delivery is more useful than the extra speed but in program like this project where speed is crucial this can cause noticeable lag in a scale that is harmful for enjoyable use of the client program.

UDP solves this issue by not having any checks whatsoever. Program sends the data and after that pays it no heed. If packet is lost there's no response to it from either server or client so there's no time spent on waiting for acknowledgements and retransmission. However this requires some sort of reliability checks to be done in one way or another. One solution would be to implement lighter acknowledgement system of your own top of UDP. For some data you don't need to be so picky whether individual packet is lost or received as new one is sent faster than acknowledgement would come anyway. Position data would be example of this for example. Since position changes often constantly anyway you could just send new position and if previous packet has been lost it's going to be just bit bigger jump than before. When you are sending about 40 position packets per second losing one or two here and there aren't going to be crucial.

Yet another method would be similar to what ID Software did in Quake 3 and is extension to idea of continuously sending data but instead of sending out only certain data and implementing reliable methods for some you would send out everything that has happened

since client's last acknowledgement message [1]. This has the advantages of reliable protocols without the disadvantages of it. However it does have its own problems. As it depends on sending out whole state there's quite a lot of data to be sent which increases size of packets quite a lot. This is only compounded by the sheer number of objects in typical MMORPG has. And while splitting world into sectors (see section 3.3) helps it in turn causes issues when it comes to shifting sectors as you need somehow to keep track of which sectors need only difference updates and which needs whole package. You also need to create efficient delta compression method which adds data from more than 1 sector at the same time.

However all in all the Quake3 network model could very well provide highest efficiency in terms of network speed. Additional computational power can be compensated by optimizing algorithms and of course improving server's computational power if needed.

However for this project TCP has been chosen for protocol despite its disadvantages for simple reason. It's easy to use. No need to create own acknowledgement system. No need to create delta compression function or save state, idea for which was found far too late in development process to swap into in feasible time. UDP was considered for a while but difficulties in designing reliable method for packets scrapped that idea. In the end since TCP's speed wouldn't become issue on this scale it won out.

However it is worth pointing out that TCP doesn't prohibit even real time network games. For example the previously mentioned MMORPG World of Warcraft uses TCP for its network protocol [2]. While likely using some advanced system to ensure occasional packet hiccups won't cause issues it's still convincing proof that for MMORPG styled programs you can achieve sufficient speed with TCP. Also the Ultima Online uses TCP as well [3]. Reason why this can work lays most likely at the nature of MMORPG which is less speed-critical than for example the first person shooters. In the first person shooter you are in constant movement and difference between life and death is in fast and accurate responses. These in turn require low latency between server and client. If TCP would here lose packet and waste even less than half a second it could very well cause player to miss target and die himself instead. With MMORPG you are generally moving with less hectic speeds and combat is generally done by ordering character to attack with server determining results even without direct input from the players. Client sees this only in slight graphical hiccup in graphics and maybe slight position jump if characters have

moved. While latency between server and client would still be irritating it is not as decisive as in first person shooters for example.

3.3 Entities involved

We have two critical entities we need, the world and the characters that inhabit the world. The world needs the data needed for creating the world in 3D representation and calculating movement around there. It should also be easily extended in case the server host wants to modify the world around. For representing the world I went through several ideas. Traditional method to generate 3D terrain is through height maps which are essentially 2d images where colour of the pixel represents height at that point. For a long time I considered using this but in the end I ran into few issues.

For one I never got into situation where I was able to alter the height in scale I wanted, though trying to use paint as graphic program probably didn't help, and instead got wildly altering distances. Secondly I started to worry about scale available. As height map uses one byte to store height value there's only 256 values. I started to wonder whether that would be enough for world where you could go way up and way down. Though you can increase multiplier for height which increases total height it's also going to result in more blocky terrain.

There's also issue of being pretty fixed to those settings. If your height multiplier is 2 then you can't have height of 1.5 at certain point. Again blocky terrain issue roars its head. But main reason I ended up discarding the idea was textures as I never figured a way to get it work. The way textures work with openGL is that they are stretched to fit the surface. Since there's upper limit to texture size, and big textures eat tons of memory anyway, end result is that you are going to have one very stretched out texture. Early attempts gave me terrain that had lots of blotches in it. Though it could work for say air simulator looking terrain from high above it would look decidedly odd for MMORPG where you are walking on ground. Believable grass for example just doesn't happen and roads would be big trouble. Line that is just 2-3 pixels wide is stretched out to be very wide flat colour and due to how narrow line was adding other shades isn't going to work either.

Now there is way to get around this, sort of, by having repeatable texture. Essentially you have texture which is repeated every once in a while. So rather than have map that is 1024

unit wide covered by 256 unit wide texture you would have it repeat 4 times in a row. This removes stretching and allows much more conceivable ground texture. But then how to implement roads? They can't be put in regular texture as we would have lots of repeating road parts then. Good if we want checkerboard, bad if we want actual roads. If we use multi-texturing and put roads in second layer we hit the issue with stretching again. Somehow we would need to be able to define where the road goes and use repeating texture for that area.

Eventually though, I settled for system that is more like early role-playing games where map composed of tiles. For example early Ultima games had tile based map where world was composed of smallish collection of different tiles which when grouped together formed big varied and interesting worlds. If that worked then why couldn't it work here?

So we would define terrain tile. That needs location in X, Y and Z axis plus vertex, normal and texture values. Since location isn't tile based it makes sense to separate the two so there would be two objects involved. First up we have the terrain piece itself which holds the necessary data for creating 3D model of the terrain piece. This I call simply terrain and they needs to be stored somewhere where separate terrain tiles can get access to. Then we have the terrain tile which has reference to one of these models, suitable texture which is also shared between terrain tiles and location data. X and Z, which are the width and depth if we imagine to look map from above, we can generate runtime from piece size and index but the height (Y) we need to store there as well. Final important data we need is direction which basically is multiplier in 90 degrees. This allows us to use same terrain tile/texture combination to represent all 4 angle possibilities without requiring new class. Other than 90 degree angles don't work due to tiles being squares.

There's also currently two redundant pieces of data that won't be used in future. Specifically passable (whether you can enter the tile or not) and walk height (which would have been used to determine height characters are standing there). Both will eventually be replaced by old fashioned collision detection which is much more accurate system though harder to implement.

World object itself will contain map of terrain tiles. However one big map of full world is inconvenient and memory hog as well. Server needs whole map in memory but for client we don't need more than immediate surroundings. And while server needs whole map it

also needs to split the characters into smaller groups which I call ‘zones’ or ‘sectors’. With this in mind world is split into number of sectors which are in size server host determines. In the example configuration each sector is 32 tiles wide and deep with the world composed of 36 sectors in square formation. So the final design for the world would be world which is divided into sectors which each are composed of terrain tiles which refers to terrain models.

For characters we need to separate two types of characters. There are the player characters which are controlled by players and non-player characters. Both the server and client handle them somewhat differently when it comes to distinction but overall most of the functionality for these two is the same. Therefore we make character to be parent object from which player characters are extended from.

On the server difference is not only that players are controlled by the clients but also that player characters needs to store slightly different information. The non-player characters don’t really need all the same skills as player characters do for example since there’s no need for them to be doing blacksmithing same way as players do. Either blacksmith sells item in question or he does not for example. Nor are non-player characters going to be lock picking doors. And then there’s the fact they are player characters causes significant additions as the network requirements must be considered. The player class also has username of client in store to prevent requirement of client to send username every time it is needed.

For client the difference comes mainly in that other player characters and non-player characters act in same way as far as client is concerned. There are only two types of characters as far as client sees it. Character controlled by this client and those which aren’t and the difference comes in information character knows about and some added functionality. For other characters than the client controls there’s no need for skill levels, specific health and so on. You don’t have access to them so you don’t get them. So character class has only bare minimum with player class extending from there to contain information they need.

3.4 Communication model

Client and server communicate with each others by sending messages to each other. These messages are simply strings in pre-defined format. Messages are themselves numbers like login command is 1 and so on. This is followed by parameters, if any, grouped by delimiter (currently / character but needs to be replaced by some less common character as this would mess with say chat messages where client types /. To make debugging easier simple ASCII letter was used here however). Different messages are divided by another delimiter, this time * though same issue applies here as well.

As the system uses TCP we don't need to consider acknowledgement messages so we have just one message for one purpose. Bare minimum what we need to be able to do with communication system required total of 30 messages. Communication flow begins with client sending join message. Parameters here are user name and password. This is either replied with no connections message, server has reached its maximum capacity, login failed message, user name and/or password were incorrect, or message giving names of all the characters clients account has. After receiving selection server finally sends out all the data required to start the game and create the character. Once both sides are ready the game can begin properly.

While logged onto the server both client and server send each other message when something interesting happens. Client for example sends how it wants to move when player changes direction or speed. If he wants to use item he would send use item message. Similarly every time character, whether some player or non-player character, moves the new position is sent to the clients who are to be told of it. When new creature is spawned clients are notified.

4 Overview of server

Server has two functions. Creating and running the world as well as providing access to this world to the clients who wish to enter it. It's divided into two parts. There's the communication part that listens for clients, takes note of their commands and responds to them. Also there's action part in which main activity happens. In there characters are moved, new creatures (non-player controlled characters) are created, combat would in theory happen and so on.

4.1 Server class

This class has several jobs and as such it is quite a large. It holds out not only the world but also characters in both map container containing all of them but also vector containers which contain another pointer to characters based on which zone they are in. This double storage is to give quick and easy access based on whether there's need to access one specific characters identified by ID or by accessing all characters that are in certain zone. If only one method would be used it would require either search through all sectors for specific character or search through map all characters and checking in which sector it happens to be. While this leads to bit of additional code to ensure there's no duplicate pointer left to character who has been deleted, and ergo leaving program crashing double memory deallocation issue potential, its small code and small price to pay for such a flexibility.

`Server` class also handles the network connection, though splitting those into separate class from which `server` class is derived could be good idea allowing network functionality be used easily in other programs as well, and contains data for spawn points and other data like items once they would be added. In other words this class contains access in one way or another to pretty much everything server has, most with as direct access as possible for sake of speed. This has resulted in rather large code which has forced splitting source code to multiple separate files organized by their functions.

4.1.1 Creature & player

`Creature` class contains all the data required to represent any creature, from a basic dog to a human to a mighty dragon. This works as a base class from which player class is derived as they are mainly identical with players just requiring bit of alternative information and the network code.

4.1.2 World, sector and terrain tiles

`World` class serves 2 functions. One is to keep list of terrain pieces. These are essentially same as client has except there's no need to keep track of textures. This data would be used only for collision detection when characters move around. Settings are loaded from world files which simply determine how many sectors world has in width and depth and how many terrain tiles each sector has.

Upon creation of `world` class instance it reads world file that was specified and starts to generate sectors. The sector files are named on X/Z(width, depth) co-ordinates. Due to openGL co-ordinate system 0,0 sector is on top-left corner when you look down to map. Sector class in turn reads sector file based on position given and reads data it needs to create sector from there. Each sector file (extension .sec) is made of 5 numbers for each terrain tile. Terrain tile object index, texture index, direction which is used so that same tile/texture combination can be used to face north, west, south and east. There's also objects height which determines height in which tile exists which is used in client for drawing and in both for collision check which would be used to determine height characters stand upon.

Each `terrain tile` in sector is composed of unique data, height, direction and so on, and common data which are vertex, uv and normal co-ordinates. Since common data is shared by many tiles they have been separated into separate class. `World` class in creation creates what could be called library of these terrain piles. Individual terrain tiles then have pointer to one of these. Advantage of this lies in reduced memory usage. By having only one copy of common data and tiles having just a pointer to object, which takes minimal amount of memory, we have cut down memory usage a lot. For example basic flat ground requires 2 polygons. That's 6 vertex. Each vertex has 3 vertex co-ordinates (x, y, z), 2 uv co-ordinates and 3 normal co-ordinates. That's 48 float values. In the basic world of

project there are 36 sectors each composed of 1024 tiles. If we would store same data for every tile we would have 1,572,864 float values. Each float is 4 bytes so we can quickly calculate that it would take over 7Mb of memory just to store all the data needed. Though you can cut down lots of this by omitting uv and normal values which aren't needed in server it would still leave client side. Also as will become apparent on client program there's another reason for avoiding this which affects performance. And finally this is just for simple world with flat ground. Add more complex terrain and we have lot more wasted. Now in comparison storing it by pointer requires just 144kt, 36864 pointers, which are same size as float generally, and 48 float values for terrain tile itself. And what's more the amount of vertex data we store is much less which becomes crucial when it comes to client and drawing the world.

4.2 Server operation

Server begins by loading up setting up SDL and loading up all the data it requires. These are loaded using servers configuration file which includes path to world data, creature data and other data as well as zone membership distance. After these are loaded the main process begins. Server works in 3 stages. First it listens network for any activity and handles it. Then it performs main activity. Finally it responds clients.

Network reading is simple enough. Read once every round. When it comes to main activity there's another issue to consider. If we would just do activity as fast as we can once per loop we would have rather bizarre effect where speed of activity like movement is dependant on server speed. In other words if it takes five seconds to reach place A from place B it could take seven seconds if server is unusually slow or 1 second if it's very fast. This obviously is detrimental to suspension of disbelief and game enjoyment. Therefore we need to take activity happen according to time that has passed since loop has last time been performed.

Basic idea could be simply limiting how often it's performed, for example 60 times per second. This however fixes only too fast server. If server runs up slower it does nothing to alleviate the issue. So let's consider different type of events there are and how they need to be handled.

First we have the ones that happen constantly but at speed that is dependant on time. For this most obvious example is movement itself. The second type is something that happens when sufficient time has happened. Examples for these would be attacks and spells which don't happen instantly but after some delay.

Another issue is that simply checking through everything is going to take a time. Though looping through 10 objects isn't going to take all that long it's altogether different if you try to go through 4000 dozens of times per second. Therefore it would be more effective if we limit the times we even loop through the system.

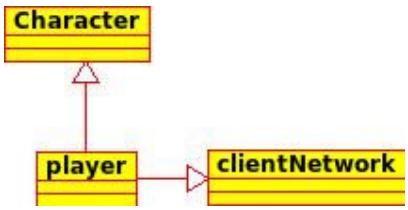
There are likely many alternative ways to do this but how I opted to do this is through logic steps. Basically there are X number of logic steps in each second. The number is determined by constant which represents time difference between each logic step. Example server has 17 for the difference which results in roughly 58 logic steps per second. Ideally you want to configure this so that there really is 1 loop for each logic steps but if server misses one due to overload it won't have critical effect due to how it works. The server keeps track of time that has passed in for loop and the moment logic steps equal or exceed one it will start internal logic. It will send how many logic steps ended up being processed (1, 2, 3 or more…). For those which use spent time will get logic steps done as parameter which is used as multiplier there when calculating activity. For attacks and other such activity we do it several times in a loop.

Above system doesn't quite create same system as if server would run without slowdowns but shouldn't go totally haywire. Technically it is possible that server slows down enough that there's several logic steps to be done and due to moving happening in one bunch character moving outside combat range before combats are done while without slowdown attack would have still been range by the time attack is done. However doing movement step by step would increase time spent within loops due to doubling, tripling or even quadrupling the amount of calculations done. In practice the issue is going to be so rare that it's not worth worrying over. If this becomes serious issue then the server is underpowered when compared to number of clients it is trying to support at the same time so either host should increase the calculation power in the server or the length between logic steps.

4.3 Server network model

Server network needs to be able to deal with multiple clients at the same time and so first decision was how to store the sockets, which TCP uses for handling connection between multiple different clients, in such a way I could quickly determine which client is the one I'm talking with. Initial thought was vector which contains all the sockets but that ran into issue of how to recognize the client that is attached to this particular socket? First solution was to use the index of socket in vector but that requires synchronizing it and player list and slightest error there would result in bucket loads of errors and isn't compatible with iterators without keeping track of index manually.

However solution came through thinking in terms of objects. Who has the socket? The player. Who is therefore obvious one to hold the socket? The player. So why not wrap network code regarding players to player class. And since C++ supports multiple inheritances I could inherit player class from character class that is used for server controlled characters and client network class which has the network code. As such player class contains all the code necessary and we have network class which can be reused elsewhere. The picture 4.1 shows the player class inheritance.



Picture 4.1. Player class inheritance diagram

With sockets location sorted out let's look at what server needs to do. It needs to a) listen for new connections. b) if new connection opens create new player class and allocate socket to it. c) read messages client sends and d) send messages to clients.

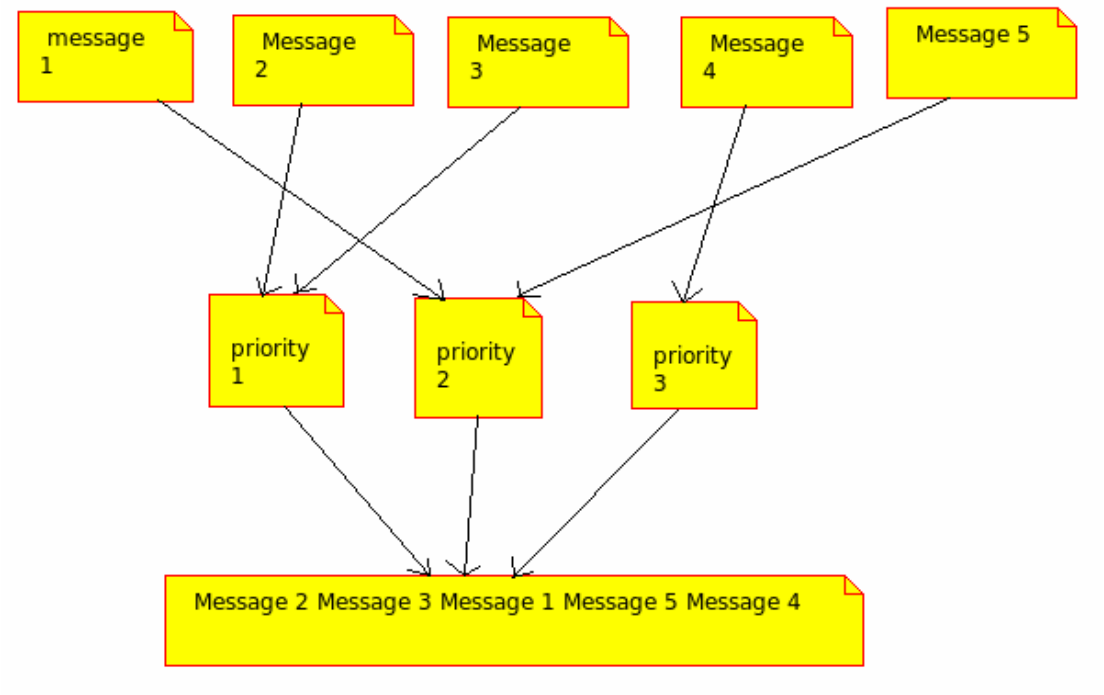
Listening for new connections is handled by trying to accept connection. If there is connection waiting we get new socket pointing to it and otherwise we get NULL socket. So every loop in the main program we create new socket and try to accept new connection to it and if we get non-NULL pointer we create new player.

TCP socket reading are by default blocking and thus attempting to read from them is going to hang the system until there are something to read. Obviously this would be very bad so

there is a solution for this in form of a set. Set is collection of sockets and you can determine in a non-blocking way whether there is activity or not. If there are you can then, again without blocking, check each individual socket whether that socket contains activity. If yes you can read it. However activity can be pretty much anything on socket including socket closing so program needs to check whether read attempt is successful or was there error which indicates that connection has been closed for whatever reason.

When it comes to replying to clients some optimization is good to consider. There is only so much data you can send to each client each second and each second needs more than one message sending so there needs to be some sort of packet size limit. For this I came up with 3 priority message queues. Rather than sending messages as system generates they are allocated to three priority queues which could be named best as normal, critical and supplemental. Critical messages contain information like character position data which should always be sent as fast as possible as delays here could cause player experience to suffer. Normal messages are ones you like you get as fast as possible but ones where slight delay is less likely to cause major issue. Examples for these would be notifications about attacks on other characters than your own, especially if they miss. Finally supplemental messages are ones that could very well be very large and ones where even serious delay is not going to be issue. For example very long chat messages in middle of furious fights are not likely going to be missed if slight delay happens and they can eat bandwidth proportionally. Then when it comes to composing message we'll be sending to client we create maximum length message and start cutting messages from priority 1 (critical) queue. If resulting message is at most maximum length then paste it and delete message from queue. If it would exceed halt process and send message. Repeat for all three priorities until either all are empty or maximum length would be exceeded.

Now for example in picture 4.2 you can see 5 messages allocated to the three priorities and resulting message from there. Now if the message 4 happens to be very large it could cause the maximum length be exceeded it would be left for future rounds, possibly until activity there quiets down, and messages 1, 2, 3 and 5 would be sent ahead. With this it would be possible to determine suitable packet size for the server which neither causes too much lag nor leaves too easily messages for later rounds (potentially infinitely). This would be dependant on servers' network capabilities.

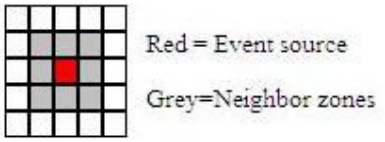


Picture 4.2 Message queue

Final consideration for the network model is the player zones which are touched before as well. As mentioned these zones are of same size as clients sector. This was done to simplify the design and limit the number of bugs that might pop up. The zones exist for one purpose. To limit who are notified by events generated by some character or event. Without any sort of limitation you would have clients notified of everything that happens on the world. If for example server has just 50 clients and 150 creatures, likely very low number as far as MMORPG go; it could easily get about 200 position updates each time they are sent. If each message is then assumed to be 15 characters long we have almost 3 kilobytes worth of position data send on one update alone. If we assume 30 times per second that's almost 100 kilobytes per second. This would end up causing hideous lag even with just basic activity by all characters. Obviously this is not a good idea. And besides there's hardly any need to know that an orc on the other side of the world got hit by a sword in a first place.

Zones fix this issue. Rather than telling everybody about an event we only tell nearby people. Specifically characters in zone you are in and characters in zones which are within certain amount of zones away. In example configuration distance is 1 so apart from

characters in zone event is triggered also the characters in neighbor zones. Picture 4.3 demonstrates example server's zone system.



Picture 4.3. Server zone event demonstration.

Here event was triggered on red zone. When it comes to reporting to players only those in either red or grey zones would be notified about the event. This way, barring overcrowding in certain sector, the amount of messages sent to each client each second should stay manageable.

5 Overview of client program

Client was the more complex program out of the two made. Where server only altered data client has to show the result to the user. This added additional issues of the openGL programming to the mix of issues to study.

Another major aspect was the graphical user interface which is needed for different menu screens, inventory, information screens and buttons to name a few. There are existing 3rd party libraries like Cegui, Turska or GLUI but for reasons of practice I opted to do it myself.

Network model is notably simpler than server has as client doesn't need to handle more than one connection and bandwidth is less likely going to become issue as amount of data to be sent is limited only related to clients own character.

5.1 Client class

Much like server class the client class is in essence orchestrating three different elements to work together. There is the GUI element which handles input and output between client and the user. There is the 3D graphical engine which handles showing the world and its inhabitants. And there is the network code which handles communication between client and the server.

Overall structure is pretty similar to the server. We have the `world` which contains `sectors` which are composed of `tiles` which refers to `terrain models`. We have `creatures`, though this time other player characters and non-player characters are lumped together with only clients own character separated, in overall list and another reference to them in individual sectors. Here reason for having them in individual sectors isn't anything to do with finding right character quickly or with sending messages to each other. Instead another important performance reason comes here in play and that is performance of the 3D engine. While current 3D graphic cards are efficient in limiting number of polygons actually drawn too much stuff drawn is still going to become an issue when it comes to efficiency if for nothing else than due to number of function calls that are done when the scene is drawn.

5.1.1 Character

Character implements functionality to create, draw and alter characters of the world. Most important functionality is drawing and moving characters around the world as these two are the most commonly used features and without which rest of the features would be pretty useless.

Character has quite a few variables but the three vector containers are probably the most important ones as these contains access to the 3D models that will eventually be drawn to the viewer. The reason there are three containers is because character can have different amount of equipment over him. The vector `bodyPartsNaked` is essentially character when he wears no equipment so would be shown in underwear's and that's it. Next vector bodyPartsEquipped shows the model when he's wearing something (say chainmail tunic over his chest). When drawing we go through `bodyPartsEquipped` container and if index isn't NULL draw it and if it's NULL draw equivalent model from `bodyPartsNaked` container. Naturally these two must be synced so index 0 on one matches index 0 on other. And both need to be of same size. The third container contains equipment that is drawn always over the character but without any other model replacing them if they aren't worn. These would be stuff like weapons and quivers. Weapons would be here, rather than separate hand model, to allow same weapon to be used in every character regardless of who wears it. It also reduces texture/model combinations as there's no need to have sword wielded by hand covered in ring mail or naked hand for example.

5.1.2 Player

Player is the representation of clients own character. It extends itself from character class which provides most of the functionality required. Where player class differs is that it contains the skills and stats of the client's character. These are information you don't see from other characters so having them on character class is just wasting memory.

5.1.3 World, sector and terrain pieces

Overall these classes are nearly identical to the class in server. Differences between terrain pieces are that in client they are drawn while server has no such need and ergo lacks the functions. `Sector` similarly contains ability to draw but it also keeps track of

players/characters other than clients own player who are located in that sector. This is done for purposes of limiting characters drawn each draw time. `World` is however where biggest number of changes have happened. In the server you have to have entire world at your fingertips all the time. Client in the meanwhile is really concerned about immediate surroundings of the player and doesn't really care about the other side of the world. While in server it's reasonable to assume memory capacity and world size are related to each other it would be unreasonable to force players to increase the amount of memory in their machine if they want to play in very large world which could quickly become necessary since the terrain data can soak up quite a lot of memory in a particularly large world. Solution to this is therefore to keep only sectors in immediate vicinity in the memory dropping and loading sectors as needed.

Updating happens in pretty crude manner as old ones are simply deleted and new ones loaded up based on new location. Original plan was to drop the zone which goes out of range, shift existing ones and load up new ones but as code doing that became longer and longer with plenty of errors I started to doubt whether it is even worth it. The shift happens pretty rarely and didn't seem to cause any noticeable drop in performance anyway. The minuscule performance increase isn't worth it compared to simpler bug free solution which won't cause any surprises in the future. And the "advanced" way isn't necessarily that much slower as it required quite a bit of calculations around and plenty of pointer shifts.

5.2 Graphical User Interface

Definitions regarding GUI:

- Core element: Group of elements for some purpose like the main window.

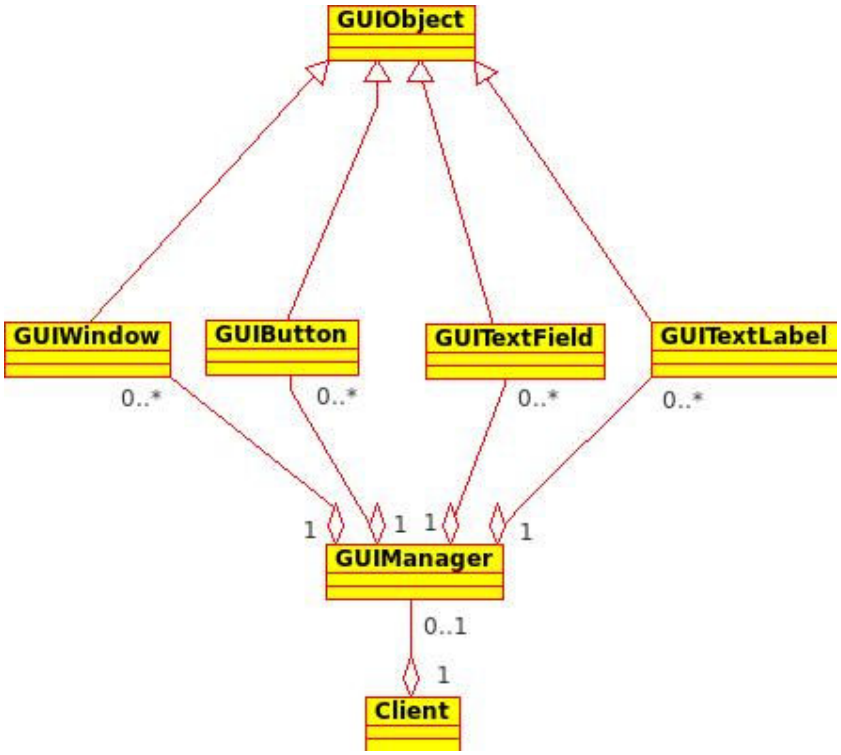
- Parent element: Element that has no parent itself and often contains children of it own. Each core element can contain only one parent element.

- Child element: Element that is part of parent element. Is located somewhere within parent element and is shown only when parent is shown.

The GUI elements are all derived from `GUIObject` class which contains common interface for GUI activity like drawing, checking whether point is within elements area and so on. The GUIObject provides access for many of the properties element has but large

number of methods like draw, move and so on needs to be implemented in each new element. This allows each element to react appropriately to different activity according to its type.

For added flexibility you can also send messages (which are essentially integer numbers) with parameters. This reduces number of functions that the interface needs. For example if you want to get the text fields text you can send TXT_GET_MESSAGE rather than needing specific function to be defined in `GUIObject` and implemented in other elements as empty function. Any return parameters are provided in string.



Picture 5.1. GUI classes and their relationships

Text printing in the GUI is handled with the library called ftgl. To provide access to same instance of ftgl class I have opted to use singleton design pattern. This pattern is used when you need to ensure there's maximum of one instance of any class available. This is done by having within class static instance of itself, hiding constructor from public use and providing access to it only through separate method which first checks whether instance already exists and if so provide access to it instead of creating new instance. [8]

Benefit of this is that you can have global access to specific instance without using global variance, though many consider singleton pattern to be just a variation of global variances, instead of having to provide instance in parameters which can result in long lists of parameters. Pushing parameters to stack is also going to take a bit of time as well.

But with just one instance available I can initialize the ftgl instance and set up fonts which are then available everywhere. Another alternative would be creating new instance of ftgl but as this requires loading up font files either I would need to use global variable to path for font file coming back to issue of global variables, hardcode font file to every place text is printed which is worst solution possible as well as takes extra CPU time when class is created and initialized over and over again. Or I could pass path to font file in parameters but here we would simply be switching one parameter to another.

Overall I didn't find sufficient amount of drawbacks compared to advantages to avoid singleton pattern. Though it can be used badly I feel it's the most appropriate solution to issue at hand. Pretty much only decent alternative is to provide pointer to instance for every class, and instances of those classes specifically, that needs it but this would result in larger number of pointers which are rather dangerous and therefore something I tried to avoid where possible.

5.2.1 Different elements

Complete GUI system requires several elements to provide all the tools required for modern user interface. Currently bare minimum is implemented with windows, buttons, text labels and text fields. These are the elements most of the interface will be composed of.

Window is familiar element for anybody who has used computers before. Rectangular area which can hold other elements and be dragged around.

Buttons are another familiar element that provides way for user to interact with the game. Unfortunately there is no animation when user clicks the button so there's no visual confirmation to the event which is something that should be worked upon.

Text label is element which allows drawing text inside the user interface. Pretty much any text user can't alter are done with text labels. These include text like window/button

captions, labels and other texts you can think of. If you know id of element you can set and get text of label by sending appropriate messages to the element.

Text field is another text related element but unlike text labels these are written by the user. These are used for input fields like username and password in login screen or sending messages to chat. Also these could be used to implement books players can write like there are in Ultima Online.

In addition there are several other elements that were planned but not implemented.

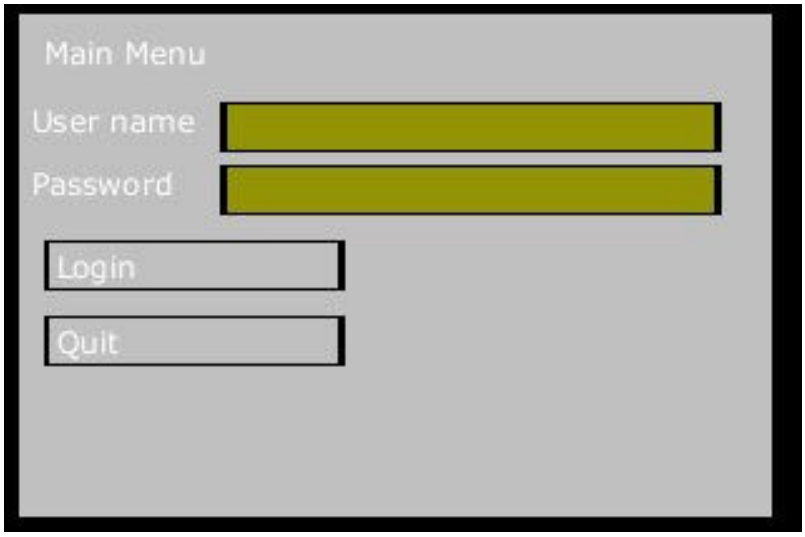
Containers would be in many ways like windows except they can accept new children elements which would be `GUIItem` type. This in essence implements graphical representation of inventory. Some items would open up new containers giving containers within containers nested effect, again much like in Ultima Online.

Placeholder would be another element that accepts new child elements. This one would accept both items and buttons. This way you could drag button and drop it into placeholder which would create copy of button to the placeholder which exists until new element is dropped. This would allow user to customize his interface by giving large amount of placeholders to the screen where he could drag commonly required buttons and items like healing potions. This idea was nicked from World of Warcraft.

Frames create sub-windows within window allowing same window to show different elements based on which frame is selected. Different frames could be chosen from the top of window where would be what appears to be buttons. Each frames child elements would be stored within separate vector containment and all handling would be done through vector that represents current frame.

List box is another very common element and one that should definitely be implemented. However the selection process has been somewhat of a problem. Biggest issue is how to draw and how to highlight the choice mouse is hovering over. This is in part due to limitation of the GUI engine which doesn’t have feature to check where mouse hovers without mouse clicks as I feared it would drain too much CPU cycle to check it every time mouse moves.

Picture 5.2. Different GUI elements as seen in the Ultima Online.



Picture 5.3. Window, text labels, buttons and text fields in the project.

5.2.2 GUIManager – heart and soul

GUIManager is the engine that runs whole GUI. Its job is to create the elements, manage them, destroy them and perform event handling directly. Client itself is only notified of events through messages. These steps detach the GUI from the client allowing it to be used for example in other applications or even swap the GUI library to new one as long as it follows same interface. It also provides higher layer of security as client has less ways to bug the code badly enough to crash the program. This comes especially in form of hiding the pointers altogether from the client program.

The GUI system creates core elements from .gui files which are plain text files providing parameters for all the elements within that core element (See appendix 1 for the file format). For example the main screen would be such a core element with window itself, buttons, text and so on being child elements. All elements, whether core or child, are loaded in master list which is used for sending messages to individual elements. Pointer to the core element is then pushed into front of list container. Childs of elements within core elements are not pushed to this but instead pointers to them are within each child's parent. Due to the high number of pointers used it was necessary to ensure client wouldn't have any access to these to prevent accidental crashes. Also several rules have been followed.

- Elements are created through `GUIManager` only.

- `GUIManager` alone destroys objects. Others do not destroy these. Doing so causes invariably crash. Therefore when object is deleted the pointers to child are made to NULL pointers, not deleted.

- Same applies to any other pointer there might be. You use it and once you don't need it you make it NULL pointer if you are closing down or you don't touch them at all. Examples of these would be parameters of message which are delivered as char array. You can read it but you shouldn't use delete[] command for it. If you do you must also set it as NULL or whole thing is going to crash the program almost certainly when same pointer is deleted second time.

As specified there are two types of lists within `GUIManager`. There's the master list containing all elements that have been used so far and there's the root list containing all the parent elements currently visible in the client. The master list is used when client wants to accept element, identified with id number, for either getting specified id of child element from specified element or for sending messages for them. Root list meanwhile is used for drawing visible GUI elements and as such this list is altered when new elements are opened or closed. It's also used to determine whether input event like mouse click affects visible elements or not. For the client this happens automatically behind the scenes and is notified of the events only through messages.

To use the GUI library there's few simple steps that must be followed. First you need to create instance of `GUIManager`. Secondly you must hook your client to the `GUIManager` so that the manager knows where to send messages. Finally client should notify of the element id for user input field if any. Client must also inherit `GUIObject` class as this provides the interface needed for the manager. From this interface client needs to implement following functions to handle them appropriately: `handleKeyBoardDown`, `handleKeyBoardUp`, and `handleGUIMessage`. Finally during the applications main loop you should call eventHandling and `drawGUI` functions or the GUI does nothing. With these steps done GUI itself is running and sending messages of events to the client.

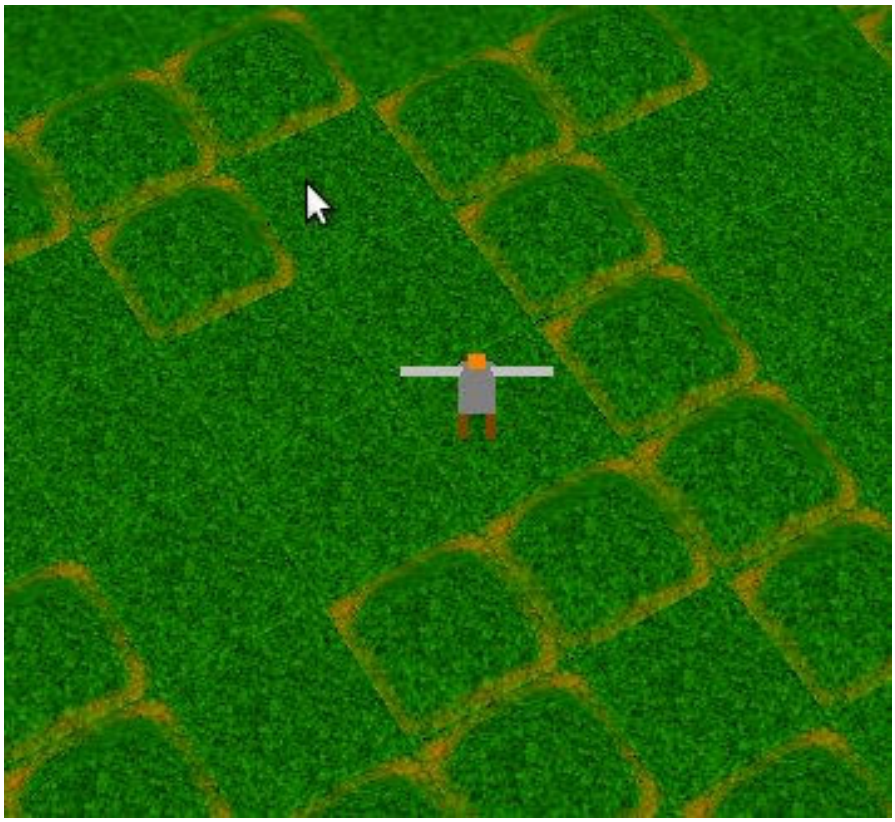
5.2.3 Message system

The GUI engine has two sorts of messages. There are event messages which inform client of events related to GUI (including main window related events) and messages which program sends to different elements. Event related messages are the kind of like "button was clicked" or "keyboard pressed". Most of these goes to the client program that has hooked itself with the manager but keyboard events are handled bit differently. Since the text fields also depend on keyboard input rather than sending keyboard events straight to client which sends to appropriate text field keyboard handlers are hooked up separately. When text field is selected it is hooked as keyboard handler and all future keyboard events are directed to that entity. Main client is hooked up as keyboard handler again when pretty much any other element which is not text field is selected with mouse.

For the message system to work the main client must implement `GUIObject` class though most of the functions don't need to be implemented (like drawing, drag and so on) as they won't be called from the manager class.

5.3 Graphic engine

Major design issue to solve in graphic engine is the file format to be used. Since models would be designed with free program called Blender it had to be something I can get from there. Unfortunately many of the file formats were rather cryptic and with too many features for my needs. Therefore I decided to use Wavefronts obj file format as a basis for my own file format. The .obj file format is nearly enough but not quite. There are some redundant data I don't need and animation is done inconveniently by storing each frame to separate files resulting in duplicate stuff. I also need some base data set up regarding animation. Therefore I set up my own file format which I dubbed eXtended Object File or .xof as the file extension.



Picture 5.4. In-game picture of the project.

5.3.1 Extended object file format

What follows is basic description of the extended object file format with consideration of how it affects performance.

File starts with basic data for animation which is the minimum and maximum frame for each state and whether state loops or not. These are followed by how many polygons, vertexes and so on object has followed by list of each polygon. Each polygon line defines 3 vertexes in vertex/uv index pairs. These are followed by individual data for each frame. How many different normal co-ordinates there are, vertex co-ordinates for each vertex index, in same order for each frame, and finally which normal co-ordinate goes to each polygon.

As can be seen the format is pretty simple which makes dealing with easy. I have made small converter which turns blender's .obj exporter result into .xof file minus animation base data. However there's serious problem with the format as such. The fact it is simple results mainly in how it deals with animation which is by storing full data for every frame separately. This leads to big memory waste. For rough estimation the leg object has currently 132 polygons, 82 vertexes, 24 uv co-ordinates and 49 frames. Each frame has about 23 normal co-ordinates. This leads to over 13,000 float values or 52 kt worth of them. While not issue on project of this size it becomes quickly issue when models become more sophisticated with bigger polygon counts and with more animation. Merely walking took 48 frames. This leaves 10 more states which need to be animated.

As the data should be loaded to graphic cards own memory for performance we can make some quick estimations. There is 512Mb worth of memory in these days on graphic cards generally so let's assume that's how much we have worth it. That's worth over 10,000 objects like that. Not too bad. Let's then assume each object has say 3 times as many vertex data and so on with 48 frames per state. This leads to memory consumption of over 1,000,000 float values or 4 Mb. Much worse situation! This leads us to mere 130 objects that will fit into the graphic cards memory. And this doesn't take into account terrain data or textures that we would also like to include. Clearly this is just not practical system but until I can create better one will do for the purposes of demonstration program.

As mentioned data will be loaded into graphic cards internal memory in as large amount as possible. Object is converted into openGL vertex arrays which are among fastest ways to

draw in openGL and store arrays into the graphic cards if there is sufficient room available. This removes the need to send up huge amount of data from main memory into graphic card every time object is drawn. As sending data to graphic card is slow, as it is defined in computer terms, this would quickly lead to bottleneck.

For speed difference let's consider following figures in table 5.1. There are frame per second rations for graphic engine without any kind of optimization, one where glBegin/glEnd pairs are replaced with vertex array and one where vertex arrays are stored into graphic cards memory. As can be seen difference between worst and best is almost double on test machine. This only gets bigger the more complicated model would be.

Version	Average FPS
Original	230
Vertex arrays	320
Vertex arrays in memory	420

Table 5.1 Speed comparison between graphic engine versions

Graphic engine is also place where singleton pattern was utilized. As mentioned rather than storing same data for every terrain tile/character part I store common data to one place and rest use common data from shared source. Rather than utilizing flat out pointer to existing instance I store index number and have class implementing the singleton pattern storing a "library" of models. There are two such libraries, one for terrain and one for character models, which both contains both the textures and the models used in the world.

5.3.2 Camera

OpenGL utilizes camera system that can be rather confusing to begin with as there is no real camera as such. The "camera" always points at 0,0,0 co-ordinates so when you draw a vertex to co-ordinates 3, 2, 0 it's 3 units to right from center of screen and 2 up. So if you have model defined with its own vertex co-ordinates how can you draw it to somewhere other than the center of screen? You do that by translating camera matrix with translate function to move the scene around. Then when you order next draw commands the draw area is elsewhere related to where you were. This system caused me more than a few

headaches having come from darkGDK (which is windows specific high level directX interface) where there was actual camera you could move around with specific orders. [5]

That handles drawing but setting up camera to specific point requires additional work. First idea I had was to simply translate scene to position where I wanted camera to work. It didn't work. Result was horrible. I never got it anywhere near where it should be. What you need to do is to move the scene with inversed transformation. When drawing objects you first move the scene and then rotate in order of X, Y and Z (order is important. Do rotations first and then move and you will have rotated the object in way you didn't want to rotate). For camera you need to rotate it first and then move and top of that you need to do it in opposite direction. In other words you want the camera be in position 5, 3 and 2 you need to translate the scene by -5, -3, and 2. Same applies for the rotation.

Additional thing to keep in mind is that openGL is based on states. Once you move scene around any additional move commands are going to move the scene based on where it was moved. NOT in original position. Unless you want to calculate translation values based on difference between last and next one you need to return to previous scene. Thankfully openGL provides method to save and restore current matrix stack so in effect you do it by saving matrix before drawing next object and restore it once you have. This way you will have common point from where you can operate from.

These features were wrapped in the `camera` class which is used to move around the world. I opted to have one camera view to character akin to viewpoint in the Ultima Online. Every time character moves the camera position is recalculated so that it's from the same angle and distance away from that point looking at it. Every time scene is drawn I then position the camera before starting to draw the scene.

Another important function `camera` class does is to calculate the area that is drawn. Ideally we would like to draw only those who are in camera view and no more. Popular method for this is the octree but that's pretty complicated and somewhat of an overkill for current engine. For free floating camera rather than one fixed to one viewpoint octree would be mandatory. Instead I have chosen to limit the amount of objects we draw to as low amount as possible. The system utilizes the world being composed of terrain tiles. We can calculate the tile character is in and from there if we calculate the area around him so

that only minimum amount of tiles is drawn we cut down most of the tiles we don't need to draw.

Unfortunately the process of calculating that area is bit trickier without calculation itself becoming bottleneck. To get it working it almost certainly needs repeated calculations which involves trigonometric functions, sine and cosine in particular, which are among slowest mathematical functions with the computers. Therefore solution is somewhat more crude but one that doesn't take too long to compute. Rather than calculating area constantly I simply select rectangle centered on player square which is just wide to cover whole screen. Very quick to calculate and isn't off by HUGE margin. Problem is more of flexibility as it's at the moment fixed with the screen resolution. Altering screen resolution changes amount of tiles drawn and therefore the size of area drawn. However this shouldn't be too much of an issue as one could provide predetermined sizes along with resolutions.

5.4 Client network model

Client network model is much simpler than the server as it doesn't need to maintain multiple connections. There's also much less data going from client to server as human can generate only so much activity on himself so there's no real need to buffer messages so instead can send messages in full constantly.

Messages client sends to server are designed to tell *what* client is going to do unlike server which tells what client *does*. On its own client does no decisions on it's own as that would provide means for cheating by sending packets that alter game state by non-legal method. For example when it comes to moving client says to server "I will be moving forward" which results in server moving character based on current location and sending position to the client. This way client never alters characters position directly. Alternative method would be client calculating new position and sending new position to the server which then inform new position to the other clients but this would require extensive cheat detections to ensure client doesn't send bogus messages, like somehow teleporting to other side of the world just like that.

Biggest issue with this dumb client model is that everything then does depend on server. Apart from server requiring more processing power the network speed can become major

bottleneck. As position data is required for clients position to change if the packets are arriving particularly slowly the client will notice a lagging effect where character moves around slowly with gaps between each position change.

There is at least one way to reduce the effect which is basically predicating where your character *will be*. Essentially it will do whatever the orders would be, like move forward when client says it will move forward, but obey server's messages when they come. This way there's no noticeable gap between moving unless clients frame rate drop noticeably. When lag happen the character moves around and once new position does arrive to the client the character position is adjusted accordingly. However this has its own lag issue as during heavy stress character can "warp" around jumping from position to position and the less in straight line movement the more noticeable the jumping around is.

The question becomes then which is better for the playing experience. Occasional freezes with knowledge that character itself is moving, just displaying the result takes time, or a jumping character. From technical point of view predicating client is reasonably easy to do since all it takes is porting same code from server to the client side so it's more of an issue about which is less disturbing to the player.

In the end I decided to not implement predicating client as it can all too easily cause potentially fatal disorientation when the character jumps around potentially in wildly different position than moments ago. From current MMORPGs Ultima Online has similar system which can be seen from characters seemingly freeze at the time of heavy lag.

6 Looking back to mistakes and how to improve

The project started out originally from reading openGL tutorials from which it grew stage by stage. This approach has cost in quite a few areas as original design parts were found inadequate and in need of replacement resulting in big pile of rewritten code thorough the development cycle. In addition to this there has been some design decisions that have been done in order to get workable system even if it's not particularly efficient for future purposes. Graphic engine and overall server system are suitable examples of later. While graphic engine works it's not viable for full game due to being memory hog. And server model while working as long as character amount stays low is going to fall apart once there becomes heavy load and has no scalability whatsoever. However it is easy to implement and test with just one computer as well. Advanced server model meanwhile is going to require major difference in whole structure of the server and preferably server farm to test with which was unviable solution.

6.1 Clients graphic engine

Never-ending struggle. Drawing static model was never issue. Drawing animation was constant struggle in figuring solution that would allow sufficient animation and one that would take neither too much processing time nor too much memory. Current system works. It's also reasonably fast though not perfect. But it eats up so much memory soon graphic cards memory would run out and then performance would drop. And if you have high number of different parts with lots of frames you could run out of main memory as well!

However there is smarter way to handle animation. One that eats up somewhat more of CPU time but allows much more natural animation that could even react to world around it. Imagine system where when you crouch wall your body lies flat against the wall. Imagine kicking big boulder with foot stopping at the boulder. With current method neither is easy to implement and requires to set up different state with more frames and more memory waste.

The solution is skeleton based animation. This system imitates human anatomy in that it is based on so called bones to which vertices are connected. Then animations rotate the bones and vertices follow. This is accomplished by assigning new value to each vertex, weight, related to all related bones. With bit of math it's then possible to calculate new vertex, uv and normal data for drawing.

However there's slight issue with this wonderful system. You need to calculate these for every vertex every frame. This can become CPU intensive. And we also hit into issue of sending new vertex data into the graphic card which can become bottleneck as has been mentioned. Once again though there exists solution. We can send bone animation to the card and then do the calculation of vertex data in the graphic cards own CPU, which is optimized for graphic related calculations as well, and thus reduce load on main CPU and remove vertex transfer stage from the process. While not necessarily as fast as current system it cuts out memory usage to minimum which would quickly become performance bottleneck in old system.

So why not use this system already? In short time. This sort of change would require lots of work as new file format supporting weighs for bones would be required, the animation itself needs to be done and top of that pixel shader language needs to be learned so that the calculations could be done within graphic cards own CPU.

6.2 Network code

TCP programming is supposedly simple but still contains few quirks that made things difficult. Main issue was from the start my decision to use line break symbol ('\n' character) as delimiter between messages. Basically code read message all the way to '\n' and then stopped. What was promptly forgotten that there was still null character ('\0') to be read which marks end of c style strings.

First issues came when I received whole bunch of garbage due to string not having '\0' character in right place. After sorting this one out by adding '\0' to the end I still received garbage. Reason was still same. I read the socket until line break character leaving the null character hanging around in the socket. Since there were still characters to be read it would then proceed to read socket reading null character. And continue reading if second message

was sent as well. Except this message was put past null character which terminates c style strings. Net result was lost messages and garbage.

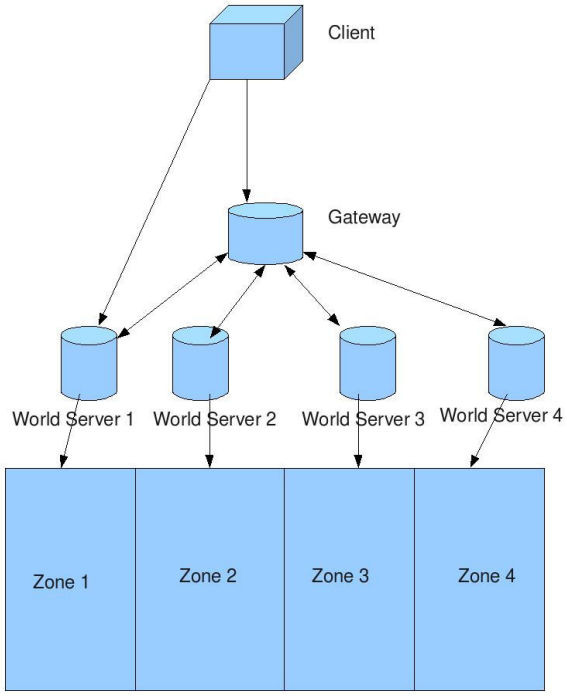
Second issue was what if line breaks are required in the message. Not needed often but if you need to send in contents of say a book? This would require special characters to represent special characters or some alternative method. Sending each line as separate message from example would be one alternative.

Bit better solution was to read until you get null character in the end. This got rid of garbage strings which could have random effects and allow '\n' characters to be added into the message without issues. However this still isn't perfect solution as it depends on pre-determined maximum length for the message. It was not until I split up messages into two parts, one consisting of length of message and the message itself that I got it "right".

6.3 Extending for large scale model

If this project would be extended for large scale action it would require some not so insubstantial changes to the way the server operates. Currently there is just one server which is supposed to handle all the clients. Though the server zone system splits up players around a bit which in turn helps when processing events it still leaves potentially hundreds of clients to handle. So what we should do is split up the players between multiple servers. This way the load individual client causes can be distributed resulting in smoother server operation.

First idea I got for this was essentially zones extended. One server would handle small slice of the world and splitting that part then into number of zones much the same way as the basic server here does. When it comes to zones which lays near the edge the servers would need to communicate with each other, preferably through high speed private connection. This would be the biggest change between this and scaled up version. In addition there's need for new server role which could be called gateway. This would be the server where player logs into when he first connects to the game. It would also be logical place for updating players' client to the new version. After logging into game the gateway would determine which zone player is currently located at and redirect player to the appropriate server. Picture 6.1 illustrates this server model.



Picture 6.1 Server model for large scale server

Advantages here lie mainly in the simplicity of the design as each server has specific area it controls and which isn't changed while server is running. Disadvantage lie in that some servers could end up very crowed while others are running with minimal player activity. This could be migrated a bit by having non-equal sized zones so that one server could be responsible only for very crowded city while second carries around work for large mostly empty sea area for example, that would in return cause issues when it comes to determining from which server to ask for data which is going to cause another layer of complexity.

Alternative method would be servers handling specific clients regardless of where they are. Rather than controlling set amount of space with unknown amount of players it would control controlled amount of players which could be located anywhere in the world. Advantages here lie in its stability and scalability as well as avoiding wasted servers. Whereupon first method had fixed number of servers which are based on size of world with only estimation of how players group here we would have very specific limits for server count. One server can handle 100 clients at same time? Then we'll have 5 if we

expect to have 400-500 clients top at the game. Depending on how clients are spread, in other words do all go to one server until it's full or spread them around the servers so that least employed gets new client, there is very little if any wasted server process.

Disadvantage here lies in that it's very, very complex system to implement. As player interacts with other characters those characters are often going to be on different characters. This requires extensive communication between server the client is connected to and server where target character is. Also informing other characters around is going to be more complex as each server doesn't have list of characters nearby of client.

Indeed for this to work there pretty much has to be central server with which to communicate. This server would also have list of each client and where they are. Then when action in character A requires all characters near him to be informed this message would be sent to central server which relies it to all servers which handle characters to be notified who in turn relies it to specific client.

This central server would have quite a lot of clients to keep track of but in return job of this server would be essentially just a post manager relying messages back and forth. However if this server would get overpopulated then the results would be delays for each client with messages concerning other characters. Or more specifically you could be pretty sure your own position is pretty accurate but all other characters near you might or might not be in place.

Another aspect to consider is non player characters and how to deal them. Here easiest solution would probably be that each individual server is responsible for certain areas much like the first large scale system controls players in specific areas. This has the disadvantage of possible crowding around certain areas and emptiness is other resulting in some servers being crowded while others are barely empty though this could be remedied by clever world design. Alternative would be splitting non player characters to different servers same as player. This would however increase work load of central server making it even more of a bottleneck.

If one could create system where there could be more than one central server and clients could predict which central server they need to contact regarding their query, and this so that it could be easily scaled up with new servers as needed, the splitting characters to different servers evenly could be the best option but if central computer can become

overcrowded then this system would be less scalable than splitting world into smaller pieces controlled by individual servers. Though even if central server would become overcrowded it's still better system than the current single server handling whole world for two reason. For one there still would be increase in client capacity due to crowded server having less work per client to do and secondly server being under too high workload wouldn't stall individual clients as far as themselves would go. They would simply notice lag when it comes to other characters.

First method is meanwhile in itself easily scalable. Add the new server; reconfigure which servers control which areas and you have additional capacity. As long as players don't crowd in same areas you will do just fine as long as number of servers is enough for the client count. Problems begin when players do crowd up. This can become issue in cities and dungeons which are traditionally common grouping areas in MMORPG. In the World of Warcraft you can easily notice slight lag which appears when you enter a city during peak hours. In dungeons this could even be fatal as you could get killed by a monster due to the lag which is something players most definitely will hate.

This issue can be solved in part by good solid world building by giving players incentive to spread around. Varying quality between dungeons and cities could end up driving players to those places which are high quality and avoid the more boring ones. Similar care must be done in other aspects of game design. For example in World of Warcraft players can choose between different races. If one race is definitely better than others players tend to choose that race in larger quantities. If access to the cities is limited according to whose side you are, like in the World of Warcraft, this too could cause crowding in certain cities while another city is looming empty.

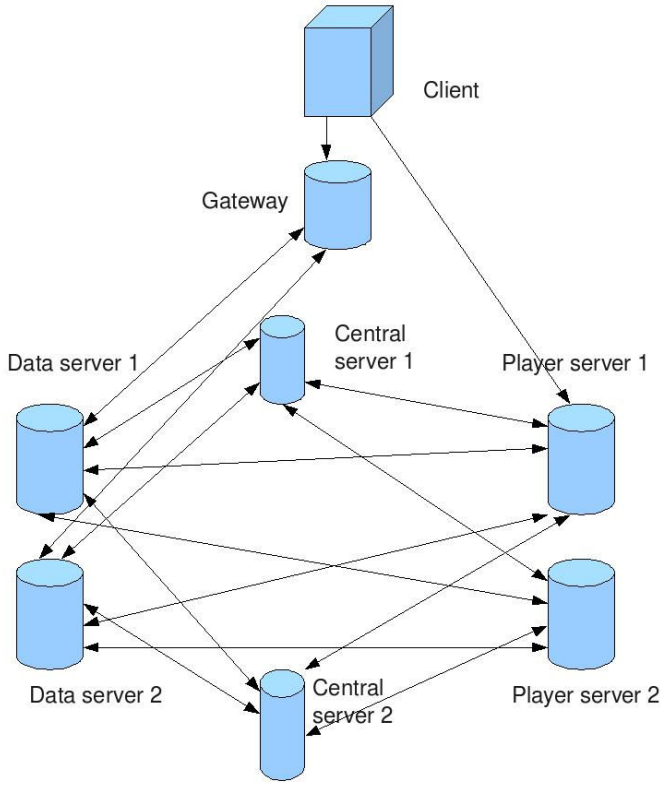
Ultimately out of the two splitting players to different servers would be preferable if one would be able to create multiple central servers. But then question is how to do that? One idea might be multiple servers each with same data which keep track of how busy they are. If they reach certain point they would pass it quickly for other servers. This would at least reduce the effect but if multiple servers are involved then message could end up travelling between them for a very long time indeed. There would also be overhead by relying message to different server even if they don't deal with message itself. So this system is unlikely going to work.

Second idea would involve creating system where server handling clients could predict which server they need to contact. One way to do that could very well be the id number each character has. There could be multiple central servers which each redirect up to certain number of clients. Each individual server would know which server handles which id, for example central server 1 handles id's 1-200, central server 2 handles 201-400 and so on, and according to that send the message to the central server handling the character who is cause of the message or the target of the action.

So in the end we have X number of individual servers and Y number of central servers. In addition we need gateway server which is where client logs in and is handed to individual server. This can be another service running on one of the central servers, preferably last one to be used, as it shouldn't be under that heavy load in any case and outside bandwidth isn't issue for central server anyway since they don't communicate with the clients directly anyway.

Finally there's one last design aspect to consider. In the above system lots of data is duplicated. For example simply the position of the position of each character needs to be duplicated in each of the central servers and maintained. So maybe it would be possible to detach this data and add yet another server type into the mix, that of data server. This data would then be accessed by other servers. This would ensure there's no need to systematically mirror data across multiple servers but does increase the internal network load due to the repeated data queries. The data read/alter calls need to be fast enough that data server itself won't become bottleneck. And again ability to scale up would be needed to avoid bottlenecks. Much as with central servers probably the best method is to divide data based on the id of the object they need. Though for reasons of safety it's probably prudent to mirror data to other servers as well and when data is altered data server then sends notification to rest of the data servers. This way if data server crashes the other servers can react to it and switch data server. Or rather send notification to one central server which then tells all servers which data server handles the job of fallen server.

The picture 6.2 demonstrates server system specified above with gateway, 2 servers handling specific players and creatures, 2 central servers and 2 data servers.



Picture 6.2 Advanced server setup

6.4 Dynamic creature spawning and removal

During the time server is running it's possible, even likely if server isn't busy like World of Warcraft, that large part of the world is empty of players with only non-player characters occupying it. Despite this the creatures eat up server resources from calculating movement and other activity non-player characters do constantly.

To reduce the server load it could therefore be useful if non-player characters are created only when there are player characters nearby. The world zones can be used to accomplish this by reasonably simple routine.

Normal method for spawning is checking repeatedly (maybe not each round but at least once per second) whether there are any spawning points that need to spawn character. This

time is dependant on when creature that was spawned it died last time. This prevents creatures spawning right after they were killed which would result in never ending flood of creatures to kill without respite. In dynamic spawning you first check whether area around it has any players or not. If there are check for spawn points as usual. If there aren't then we'll skip checking altogether.

Each sector would therefore need to contain counter for number of players it and its neighbors has. When player leaves sector the sector he left informs sectors around him that player has left it and the other sectors adjust counter appropriately. Sector to which player enters similarly informs its neighbors of new player and they adjust the counter. If ever sector finds that there are no players in it or in neighbors (in other words counter reaches 0) all creatures from the spawning points are deleted. The creatures aren't killed so the spawn point time counter isn't adjusted least leaving sector become easy way to kill the creatures.

This has benefit of reducing server load but drawback is that if the server's capacity is calculated without maximum load the servers could become seriously overloaded if players suddenly spread around the world activating each spawning point. Best way to ensure this doesn't happen is simply to scale server power enough to cope with worst case scenario in which case this system might not be that useful after all.

Another aspect dynamic spawning could be incorporated with would be splitting spawn points into several priority systems with variable levels of creatures to spawn. Idea would be that if server load becomes too much to bear lower level spawn points are disabled and higher level ones spawn harder and tougher creatures instead to compensate for reduced challenge in that area. So for example instead of cave with 4 orcs you would have 2 giants or even one dragon when server load increases. Therefore creature amount in the world adjusts itself based on server load.

7 Summary

Though basic idea for MMORPG might sound simple in practice the sheer size and complexity of them requires incredible amount of code even for basic system like one created here. Programming a MMORPG could be compared with expanding balloon as you design one feature and end up with whole lot new features that are needed for the initial design to work in a first place.

What came as a result was reasonable graphical user interface, elemental graphic engine with some work in need for animation, rough TCP based network protocol and basic structure for the server and client. By changing animation system and perhaps, if it becomes necessary, changing network system to UDP you would be well on its way toward your own MMORPG provided you have time and resources to do rest of the work which consist of actions players can do within the world and how non-player characters react.

From design point of view improved graphic engine is very likely biggest individual work left to do. While different actions players can do is on a whole huge workload they are separated into multiple smaller actions which are individually easier to implement.

Apart from MMORPGs the basic functionality could be used for number of other projects which utilizes client-network system. If one would do project that doesn't require complex 3D-animations like many 2d games many of the issues raised here would completely vanish making the development process notably simpler. This project could very easily be changed to turn based strategy game with 2d graphics and biggest issue would be the internal game logic rather than technical aspects.

References

[1] Brian Hook, “The Quake3 Networking Model”. Available at: http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking (Retrieved on 5.5.2009)

[2] Blizzard Entertainment, ”Port Information for Firewalls, Proxies and Routers”, Available at: http://us.blizzard.com/support/article.xml?articleId=21109&rhtml=true?rhtml=y (Retrieved on 5.5.2009)

[3] Electronic Arts Inc. ”Ultima Online technical support”. Available at http://support.uo.com/tech_0.html (Retrieved on 5.5.2009)

[4] ImaNewbie does Britannia. Available at http://www.imanewbie.com (Retrieved on 10.9.2009)

[5] Richard S. Wright Jr, Benjamin Lipchak, Nicholas Haemel. OpenGL SuperBibble 4th edition, Addison-Wesley. Michigan. 2007

[6] OpenGL 2.1 Reference Pages. Available at http://www.opengl.org/sdk/docs/man (Retrieved on 14.9.2009)

[7] Dave Astle, Kevin Hawkins. Beginning OpenGL Game Programming. Course Technology. Boston. 2007.

[8] Erich Gamma, Ralph Johnson, Richard Helm. Design Patterns – Elements of Reusable Object-Oriented Software. Pearson Education Limited. New Jersey. 1997.

[9] Andrew Mulholland, Teijo Hakala. Programming Multiplayer Games. Wordware Publishing. Plano, Texas. 2004.

[10] Nehe Productions. ”OpenGL Tutorials”. Available at http://nehe.gamedev.net (Retrieved on 21.9.2009)

Appendices

Appendix 1 .gui file format

Appendix 1: .gui file format

File consist of number of lines each indicating object. Format of line is:

Type parameter parameter1...parameterN. Parameters count and type depend on type of element. Types (and their parameters) are:

0 GUIWindow:
Parameters (in order): id, x, y, width, height, texture File, draggable, accept Childs, parentId

1 GUIButton:
Parameters (in order): id, x, y, width, height, texture File, draggable, parentId

2 GUITextField:
Parameters (in order): id, x, y, width, height, texture File, multiline, max length, parentId

3: GUIContainer:
Parameters (in order): id, x, y, width, height, texture File, parentID

5: GUIPlaceHolder
Parameters (in order): id, x, y, width, height, texture File, parentID

6: GUIFrames:

Parameters (in order): id, x, y, width, height, texture File, number of frames, parentID

This is followed by n number of entries specified above which each contains line for GUIWindow object which represents specific frame and line for GUITextLabel for frame caption.

7: GUITextLabel:

Parameters(in order): id, x, y, parentId
Followed by line that contains text in label

Notes on parameters:

ID: Should be set below 0 if ID isn't used to identify which object was used like text labels which aren't modified runtime etc. In this case GUI itself will allocate random ID. Only use specified ID's to objects like buttons etc where you need to access it or identify it specifically.

x, y, width, height: If below 0 then it specifies percentage of size instead. For example -40 would be 40% of screen width if put to parents width.

texture File: There can be no spaces in it.

draggable, multiline, accept Childs: 0=no, 1=yes

parentId: Smaller than 0 if element has no parents.

Max length: Determines maximum length of one line.